MICROSOFT®

TECHNICAL JOURNAL

Microsoft Confidential **Editor, Ross Garmoe**

November 7, 1984
Volume 1, Number 5

"This product will be small, fast and meet everyone's needs."

PL/I Designer

Editor's Comments - Ross Garmoe

In the last issue of the journal I discussed areas where we needed to increase our technical effort if we are maintain our position as the most influential producer of microcomputer software. These areas are:

- responsiveness to customer needs
- design methodology
- implementation techniques
- testing and verification techniques
- scheduling

The saying that 'knowledge is power' is true at Microsoft only if it is shared and contributes to the progress of the group as a whole. If 'knowledge' is considered a Microsoft group resource, several important things begin to happen:

- product quality improves because designs are reviewed
- error rates in release products are reduced
- improved practices become widespread
- past mistakes are avoided (new ones will still be allowed)
- the work environment improves

To achieve these gains we need to work on the following areas:

- transferral of design decisions
- transferral of programming decisions
- effecient transmission of 'knowledge' and 'folklore'

My experience with Microsoft indicates that very little is done to document why design decisions are made. The little documentation that does exist generally explains only the details of the selected design, not what alternatives were investigated and why they were rejected. In fact, this tendency to pretend rejected alternatives never existed is common thoughout the industry. What we must do is document the entire design process including the rejected alternatives and what are the expected advantages and disadvantages of the selected design. This will allow us to later evaluate the design process and improve our design abilities. Well documented designs also allow new members to understand the intent of the project and get up to speed in a shorter amount of time. Also, if people understand the intent and thrust of the design, there is less temptation to add extraneous or incompatible features.

Many of the current and future Microsoft products call for close interaction among the different groups. For example, Windows requires detailed cooperation among ISG, OS and SL. Other projects will require as extensive an integration of groups. Detailed design documents will ease this integration and give all of the groups a common basis for discussion. If changes in the project specification have to be made, everybody will be working from a common knowledge base.

Even after a project's design is completed and documented, there is a large amount of latitude in the programming decisions that can be made. For example, the layout of data structures, exactly how the program is modularized and the expected entry and exit conditions of the various routines. These decisions should be well documented in the program. Many people claim that one of the advantages of a higher-level language is that the code is self-documenting. This may be true if the reader of the code understands the the problem, the design of the program and remembers all of the conventions about the state of global variables on entry and exit to the routines and

the program is well written. However, if any of the above is not true, trying to understand the program is changed into trying to figure out a puzzle. Puzzle playing is not what we should be doing. We are trying to implement products that help people solve real world problems and make their life easier.

One of the documents included in this issue of the journal is the *Microsoft Software Coding and Design Principles*. You should read this and try to understand the intent of what is being said. The first reaction of many people is to become upset because the indenting style is different or the subroutine header is different. Another frequently heard complaint about programming standards is that by requiring a specific style, we are restricting the creativity of our programmers. The creative part of programming is designing a program to solve a problem. How the program is formatted is detail work and should not be considered a creative process.

What the document is really trying to say is that the way the program is formatted and commented is as least as important as how the program solves the algorithmic problem. It is essential that the program be written and formatted in a manner that maximizes the information flow from the writer to the reader. In a large organization, this requires that a uniform and coherent programming style be used.

In any organization, there is a lot of information that is fundamental knowledge and becomes part of part of the organization folklore. Examples of this are what instructions of the 286 are not to be used or how to read common idioms in C language routines. The problem with these becoming folklore is that their propagation in uneven. The way you find out this common knowledge is when somebody comes up and begins a sentence with "You idiot, everybody knows..." Preventing these tirades is one of the purposes of this journal. If you are the expert in some arcane subject such as 'What do all of the linker messages mean' and you are tired of asnwering the same questions, write an article and make everybody an expert. Also, if there are general rules that everybody needs to know, write it down and let them know.

Moving your PC-AT - Ross Garmoe

There is a problem that can occur when you move a PC-AT. If you move the machine without having the heads outside of the writable surface area of the hard disk, the surface can be damaged. Before you move the machine, you must run the diagnostics disk that was delivered with the machine and execute the option to prepare the machine for moving. To simplify the process of moving an AT, we will supply a routine for DOS called PARKHEAD which perform this function. A future XENIX 286 release will also park the heads as part of the haltsys and shutdown commands. Until you receive these routines, be sure to park the heads using the diagnostics disk.

How to Read and Write C Declarations - Ralph Ryan

One of the more confusing aspects of the C language to novice programmers is the syntax of complex declarations. Presented with something like:

```
char *(**(*f)())[10];
```

it is not obvious to everyone that this declares a pointer to a function returning a pointer to a pointer to an array of 10 pointers to char.

This issue of ColumnC will give an easy way to understand and write such declarations. The secret is to read them 'inside out'. There are a few simple rules that will be applied to declarations.

1. Start with the name and work toward both ends.
2. [ ] and ( ) (on the right) take precedence over * (on the left).
3. Parenthesis can be used to alter the natural association order.
4. The 'near' and 'far' keywords modify the thing immediately to their right.

So the scheme is to start with the name, and look to the right for [or (. Otherwise, look to the left for *. A right parenthesis says to accumulate all of the stuff inside before venturing outward.

Example 1

```
int * foo;
^   ^   ^
3   2   1

1. foo is              (rule 1)
2. a pointer to
3. an int
```

Example 2

```
int  *foo[10];
^    ^  ^  ^
4    3  1  2

1. foo is
2. an array of 10      (rule 2)
3. pointers to
4. int
```

Example 3

```
int (*foo)[10];
^   ^ ^  ^
4   2 1  3
```

1. foo is
2. a pointer to (rule 3)
3. an array of 10
4. int

Example 4

```
char far* * p;
^         ^ ^ ^
4         3 2 1
```

1. p is a
2. pointer to a
3. far pointer to (rule 4)
4. char

Looks easy? Lets try the example from the introduction:

```
char * (* * (* f)())[10];
^    ^ ^ ^  ^ ^ ^  ^
8    7 5 4  2 1 3  6
```

1. f is (rule 1)
2. a pointer to (rule 1 and 3)
3. a function returning (rule 2)
4. a pointer to
5. a pointer to
6. an array of 10 (rule 1 and 2)
7. pointers to
8. char

OK. Now everybody is an expert on reading these things. Clearly the process can be applied in reverse. Suppose we want to declare foo to be an array of 10 pointers to functions returning pointers to arrays of 5 ints.

1. foo is an
2. array of 10
3. pointers to
4. functions returning
5. pointers to
6. an array of 5
7. ints.

Doing this in a couple of steps:

```
foo[10]                     (1 and 2)
*foo[10]                    (3 : remember the precedence ?)
(*foo[10])()                (4)
*(*foo[10])()               (5 : again -- () or [] over *)
(*(*foo[10])())[5]          (6)
int (*(*foo[10])())[5]      (7)
```

And for instructive value, decomposing it again:

```
int (*(* foo[10])())[5]
    ^ ^ ^  ^   ^ ^  ^
7     5 3 1  2     4  6
```

Here's a few more to practice on (the answers are below -- don't peek):

1. w is a function returning a pointer to an array of 3 pointers to functions returning longs.
2. x is a near pointer to a far pointer to an array of 7 far pointers to near pointers to chars.
3. char far *(*far y())[10] ;
4. char far *(far *(far * near z[5])())() ;

Answers

1. long (*(*w())[3])()
2. char near * far *(far * near * x)[7];
3. y is a far function returning a pointer to an array of 10 far pointers to char.
4. z is a near array of 5 far pointers to functions returning far pointers to functions returning far pointers to char.